Bujar Sefa
CSC 21200 - BC
Professor Liu
HW #6

Overview of homework:

This homework is about creating a graph class via adjacency list and adjacency matrix. Adjacency matrix works with 2 dimensional arrays, where as adjacency list works with nodes. While their are many differences between an adjacency matrix and adjacency list (in terms of implementation of course), their main functions of add vertex, add edge, remove edge, and many more are all the same. The other part of this assignment consisted of programming breadth first and depth first as graph traversal functions. The last part of this homework consisted of coding Dijkstra's algorithm, which is an algorithm to compute the shortest distance between a start vertex to every other vertex in the graph.

Graph-Adjacency Matrix:

Header File: Sefa_Bujar_HW6_Q1.h:

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

#include <cstdlib>
#include <cassert>
#include <iostream>

using namespace std;

template<class Item>
class graph{
	public:
		graph(const size_t& init_cap = 1);
		graph(const graph<Item>& source);
		~graph();
		void operator=(const graph<Item>& souce);
		void resize(const size_t& new_size);
		void addVertex(const Item& label);
		void addEdge(const size_t& source, const size_t& target, const unsigned int&
weight);
```

```
		void removeEdge(const size_t& source, const size_t& target);
		void print();
		size_t size() const; /*Returns the number of vertices*/
		size_t numEdgesTotal(); /*const;*/
		/*NOTE total edges of specific vertex*/
		size_t numEdges(const size_t& vertex);
		bool isConnected(const size_t& source, const size_t& target); /*NOTE this can
just call numEdges and compare it to 0. */
		Item* neighbors(const size_t& vertex, size_t& size); /*Returns an 'array' of the
vertices that are connected to the given vertex.*/
		size_t getWeight(const size_t& source, const size_t& target);


	private:
		unsigned int ** edges; /*THe "weighted" edges array*/
		size_t CAPACITY;
		size_t numOfVertices; /*Basically like a count*/
		Item * labels; /*Data of each vertex*/


};


void makeHeap(int arr[], int weigh[], size_t size);
void reheapify_down(int arr[], int weigh[], size_t size);
void heapSort(int arr[], int weigh[], size_t size);
#include "Sefa_B_HW6_Q1.cpp"



#endif
```

Implementation File: Sefa_Bujar_HW6_Q1.cpp:

```
#ifndef __GRAPH_CPP__
#define __GRAPH_CPP__
#include "Sefa_B_HW6_Q1.h"
template<class Item>
graph<Item>::graph(const size_t& init_cap){
	CAPACITY = init_cap;
	edges = new unsigned int*[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		edges[i] = new unsigned int[CAPACITY];
	}
	for(size_t i = 0; i < CAPACITY; i++){
		for(size_t j = 0; j < CAPACITY; j++){
			edges[i][j] = 0;
		}
	}
	labels = new Item[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		labels[i] = Item();
	}
	numOfVertices = 0;
```

```
}

template<class Item>
graph<Item>::~graph(){
	delete labels;
	for(size_t i = 0; i < CAPACITY; i++){
		delete edges[i];
	}
	delete edges;
	numOfVertices = 0;
	CAPACITY = 0;
}

template<class Item>
graph<Item>::graph(const graph<Item>& source){
	CAPACITY = source.CAPACITY;
	edges = new unsigned int*[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		edges[i] = new unsigned int[CAPACITY];
	}
	labels = new Item[CAPACITY];
	numOfVertices = source.numOfVertices;
	for(size_t i = 0; i < CAPACITY; i++){
		for(size_t j = 0; j < CAPACITY; j++){
			edges[i][j] = source.edges[i][j];
		}
	}
	for(size_t i = 0; i < CAPACITY; i++){
		labels[i] = source.labels[i];
	}
}

template<class Item>
void graph<Item>::operator=(const graph<Item>& source){
	if(this == &source){
		return;
	}
	delete labels;
	for(size_t i = 0; i < CAPACITY; i++){
		delete edges[i];
	}
	delete edges;
	CAPACITY = source.CAPACITY;
	edges = new unsigned int*[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		edges[i] = new unsigned int[CAPACITY];
	}
	labels = new Item[CAPACITY];
	numOfVertices = source.numOfVertices;
	for(size_t i = 0; i < CAPACITY; i++){
		for(size_t j = 0; j < CAPACITY; j++){
			edges[i][j] = source.edges[i][j];
		}
```

```
	}
	for(size_t i = 0; i < CAPACITY; i++){
		labels[i] = source.labels[i];
	}
}

template<class Item>
void graph<Item>::resize(const size_t& new_size){
	if(new_size == CAPACITY)
		return;
	if(new_size < numOfVertices)
		return;
	Item * newLabels = new Item[new_size];
	for(size_t i = 0; i < new_size; i++){
		newLabels[i] = Item();
	}
	for(size_t i = 0; i < numOfVertices; i++){
		newLabels[i] = labels[i];
	}
	delete labels;
	CAPACITY = new_size;
	labels = newLabels;
}


template<class Item>
void graph<Item>::addVertex(const Item& label){
	if(!(numOfVertices < CAPACITY)){
		size_t size = CAPACITY*2;

		unsigned int ** larger = new unsigned int*[size];
		for(size_t i = 0; i < size; i++){
			larger[i] = new unsigned int[size];
		}
		for(size_t i = 0; i < size; i++){
			for(size_t j = 0; j < size; j++){
				larger[i][j] = 0;
			}
		}
		for(size_t i = 0; i < numOfVertices; i++){
			for(size_t j = 0; j<numOfVertices; j++)
				larger[i][j] = edges[i][j];
		}


		for(size_t i = 0; i < numOfVertices; i++){
			delete [] edges[i];
		}
		delete edges;

		edges  = larger;

		Item * largerLabels = new Item[size];
		for(size_t i = 0; i < size; i++){
```

```
                largerLabels[i] = Item();
            }
            for(size_t i = 0; i < numOfVertices; i++){
                largerLabels[i] = labels[i];
            }
            delete labels;
            labels = largerLabels;
            CAPACITY = size;
        }
        labels[numOfVertices] = label;
        numOfVertices++;

}

template<class Item>
void graph<Item>::addEdge(const size_t& source, const size_t& target, const unsigned int&
weight){
        assert((numOfVertices > 0) && ((source < numOfVertices) && (target < numOfVertices)));
        assert(!isConnected(source,target));
        assert(weight > 0);
        edges[source][target] = weight;
}

template<class Item>
size_t graph<Item>::numEdges(const size_t& vertex){
        assert((numOfVertices > 0) && ((vertex < numOfVertices)));
        size_t totEdge = 0;
        for(size_t i = 0; i < numOfVertices; i++){
            if(isConnected(vertex, i)){
                totEdge++;
            }
        }
        return totEdge;
}

template<class Item>
bool graph<Item>::isConnected(const size_t& source, const size_t& target){
        assert((numOfVertices > 0) && ((source < numOfVertices) && (target < numOfVertices)));
        /*cout << "NUME     "<<numEdges(source, target) << endl;*/
        return (edges[source][target] != 0);
}

template<class Item>
void graph<Item>::removeEdge(const size_t& source, const size_t& target){
        assert((numOfVertices > 0) && ((source < numOfVertices) && (target < numOfVertices)));
        assert(isConnected(source, target));
        edges[source][target] = 0;
}

template<class Item>
void graph<Item>::print(){
        /*cout << "VinP " <<numOfVertices;*/
        for(size_t i = 0; i < numOfVertices; i++){
```

```
                for(size_t j = 0; j < numOfVertices; j++){
                        cout << edges[i][j] << "\t";
                }
                cout << endl;
        }
}

/*Pre-condition, must give a size and a vertex looking for. Size will be adjusted.*/
template<class Item>
Item* graph<Item>::neighbors(const size_t& vertex, size_t& size){
        assert((vertex < numOfVertices) && (numOfVertices > 0));
        size_t numNeigh = numEdges(vertex);/*0;*/
        Item* vertexNeighbors = new Item[numNeigh];
        Item* neighBorWeights = new Item[numNeigh];
        size_t indexNeigh = 0;
        for(size_t i = 0; i < numOfVertices; i++){
                if(isConnected(vertex, i)){
                        vertexNeighbors[indexNeigh] = i;
                        neighBorWeights[indexNeigh] = edges[vertex][i];
                        ++indexNeigh;
                }
        }

        heapSort(vertexNeighbors, neighBorWeights, numNeigh);
        size = numNeigh;
        return vertexNeighbors;
}

template<class Item>
size_t graph<Item>::numEdgesTotal(){ //const{
        assert(numOfVertices>0);
        size_t totEdges = 0;
        for(size_t i = 0; i < numOfVertices; i++){
                for(size_t j = 0; j < numOfVertices; j++){
                        totEdges++;
                }
        }
        return totEdges;
}

template<class Item>
size_t graph<Item>::size() const{
        return numOfVertices;
}

template<class Item>
size_t graph<Item>::getWeight(const size_t& source, const size_t& target){
        assert(numOfVertices > 0);
        assert(source < numOfVertices && target<numOfVertices);
        return edges[source][target];
}

#endif
```

Graph-Adjacency List:

Header File: Sefa_Bujar_HW6_Q2.h:

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

#include <cstdlib>
#include <cassert>
#include <iostream>

#include <string> /*"to hold the double data type ie vertex,weight" */

#include "node2.h" /*NOTE, this was taken from the textbook. */
using namespace std;

template<class Item>
class graph{
	public:
		graph(const size_t init_cap= 2); /*init cap is to set the size of the array of
head_pointers or the vertices*/
		~graph();
		graph(const graph<Item>& source);
		void operator=(const graph<Item>& source);
		void addVertex(const Item& label);
		void addEdge(const size_t& source, const size_t& target, const size_t& weight);
		void removeEdge(const size_t& source, const size_t& target);
		size_t size() const;
		size_t numEdgesTotal();
		size_t numEdges(const size_t& vertex); /* source, const size_t& target);*/
		bool isConnected(const size_t& source, const size_t& target);
		void print();
		size_t getVertexNum(node<string> * current);
		size_t getWeightNum(node<string> * current);
		int* neighbors(const size_t& source, size_t& size);
	private:
		Item * labels;
		node<string> ** list;
		size_t numOfVertices;
		size_t CAPACITY;
};
#include "Sefa_B_HW6_Q2.cpp"
#endif
```

Implementation File: Sefa_Bujar_HW6_Q2.cpp:

```
#ifndef __GRAPH_CPP__
#define __GRAPH_CPP__
#include "Sefa_B_HW6_Q2.h"

template<class Item>
graph<Item>::graph(const size_t init_cap){
	CAPACITY = init_cap;
	labels = new Item[CAPACITY];
	for(size_t i = 0; i < CAPACITY; i++){
		labels[i] = Item();
	}

	list = new node<string>*[CAPACITY];
	numOfVertices = 0;

}

template<class Item>
graph<Item>::~graph(){
	/*Go through the list and clear out all nodes.*/
	for(size_t i =0; i< CAPACITY; i++){
		list_clear(list[i]);
	}
	/*delete the array of the list.*/
	delete [] list;
	delete labels;
	CAPACITY = 0;
	numOfVertices = 0;
}

template<class Item>
graph<Item>::graph(const graph<Item>& source){
	CAPACITY = source.CAPACITY;
	list = new node<string>*[CAPACITY];
	labels = new Item[CAPACITY];
	numOfVertices = source.numOfVertices;
	for(size_t i = 0; i < CAPACITY; i++){
		labels[i] = source.labels[i];
		node<string> * head;
		node<string> * tail;
		list_copy(source.list[i],head,tail);
		list[i] = head;
	}
}


template<class Item>
void graph<Item>::operator=(const graph<Item>& source){
	if(this == &source){
		return;
	}
	for(size_t i =0; i< CAPACITY; i++){
```

```
                list_clear(list[i]);
        }
        CAPACITY = source.CAPACITY;
        delete labels;
        list = new node<string>*[CAPACITY];
        labels = new Item[CAPACITY];
        for(size_t i = 0; i < CAPACITY; i++){
                labels[i] = Item();
        }
        numOfVertices = source.numOfVertices;
        for(size_t i = 0; i < CAPACITY; i++){
                labels[i] = source.labels[i];
                node<string> * head;
                node<string> * tail;
                list_copy(source.list[i],head,tail);
                list[i] = head;
        }
}

template<class Item>
void graph<Item>::addVertex(const Item& label){
        if(!(numOfVertices < CAPACITY)){
                size_t size = CAPACITY*2;
                Item * labelsNew = new Item[size];
                for(size_t i = 0; i < size; i++){
                        labelsNew[i] = Item();
                }
                for(size_t i = 0; i < numOfVertices; i++){
                        labelsNew[i] = labels[i];
                }
                delete labels;
                labels = labelsNew;
                node<string>** listNew = new node<string>*[size];
                for(size_t i = 0; i < numOfVertices; i++){
                        node<string> * head;
                        node<string> * tail;
                        list_copy(list[i], head, tail);
                        listNew[i] = head;
                }
                for(size_t i = 0; i < numOfVertices; i++){
                        list_clear(list[i]);
                }
                delete [] list;
                list = new node<string>*[size];
                for(size_t i = 0; i < size; i++){
                        node<string> * head;
                        node<string> * tail;
                        list_copy(listNew[i], head, tail);
                        list[i] = head;
                }
                CAPACITY = size;
        }
        string data = "" + to_string(numOfVertices) + "." + to_string(0);
        list[numOfVertices] = new node<string>(data, NULL);
```

```
	labels[numOfVertices] = label;
	++numOfVertices;
}

template<class Item>
void graph<Item>::addEdge(const size_t& source, const size_t& target, const size_t& weight){
	assert((numOfVertices>0) && ((source < numOfVertices) && (target < numOfVertices)));
	assert(!isConnected(source, target));/*Make sure these two verticies aren't already
conencted.*/
	string data = "" + to_string(target)+"." + to_string(weight); /*NOTE the target and
weight is sotred in a string. Target first then weight*/
	list_insert(list[source], data); /*insert after the head pointer the data. */
}

template<class Item>
void graph<Item>::removeEdge(const size_t& source, const size_t& target){
	assert((numOfVertices>0) && ((source < numOfVertices) && (target < numOfVertices)));
	assert(isConnected(source, target));/*Make sure the two edges are actually connected.*/

	/*Have to find the node previous to the one im looking for to use list_remove.*/
	node<string> * nodeRemove = list[source];// = list_locate(list[source], )
	node<string> * prevNode = NULL; /*lets set to current*/
	bool found = false;
	size_t valCurr = getVertexNum(nodeRemove);
	while(nodeRemove!=NULL && !found){
		if(valCurr == target){
			found = true;
		}
		else{
			prevNode = nodeRemove;
			nodeRemove = nodeRemove->link();
			if(nodeRemove!=NULL) /*NOTE NOTE NOTE. This must be checked because our
checking if at end is very poor. so precaution not to call function*/
				valCurr = getVertexNum(nodeRemove);
		}
	}
	cout << prevNode->data() << endl;
	prevNode->set_link(nodeRemove->link( ));
}



template<class Item>
void graph<Item>::print(){
	for(size_t i = 0; i < numOfVertices; i++){
		node<string> * current = list[i];
		cout << "Vertex " << getVertexNum(current) << "   Edges ";
		current = current->link();
		while(current!=NULL){
			cout << getVertexNum(current) << " ";
			current = current->link();
		}
		cout << endl;
	}
```

```
}

template<class Item>
size_t graph<Item>::size() const {
	return numOfVertices;
}

template<class Item>
bool graph<Item>::isConnected(const size_t& source, const size_t& target){
	assert(numOfVertices> 0);
	assert(source < numOfVertices);
	assert(target < numOfVertices);
	bool isFound = false;
	node<string> * current = list[source];
	size_t currentVal = getVertexNum(current);
	while((current !=NULL) && (!isFound)){
		if(currentVal == target)
			isFound = true;
		current = current->link();
		if(current!=NULL) /*NOTE NOTE NOTE. This must be checked because our checking if
at end is very poor. so precaution not to call function*/
			currentVal = getVertexNum(current);
	}
	return isFound;

}

template<class Item>
size_t graph<Item>::getVertexNum(node<string> * current){
	assert(current!=NULL);
	return stoi((current->data()).substr(0, (current->data()).find(".")));
}

template<class Item>
size_t graph<Item>::getWeightNum(node<string> * current){
	assert(current!=NULL);
	return stoi((current->data()).substr((current->data()).find(".")+1));
}

template<class Item>
int * graph<Item>::neighbors(const size_t& source, size_t& size){
	assert(source < numOfVertices); /*Make sure we have in list. */
	size_t length = list_length(list[source]);/*Get the space we need*/
	cout << "LEN "<<length << endl;
	int * neighborHood = new int[length-1];
	size_t index = 0;

	node<string> * current = list[source];
	current = current->link();
	while(current!=NULL){
		neighborHood[index] = getVertexNum(current);
		index++;
		current = current->link();
	}
```

```
	size = length-1;
	return neighborHood;
}

template<class Item>
size_t graph<Item>::numEdges(const size_t& vertex){
	assert((numOfVertices > 0)&& (vertex < numOfVertices));
	size_t totEdge = list_length(list[vertex])-1;
	return totEdge;
}

template<class Item>
size_t graph<Item>::numEdgesTotal(){
	assert(numOfVertices > 0);
	size_t totEdges = 0;
	for(size_t i = 0; i < numOfVertices; i++){
		totEdges+= (list_length(list[i])-1);
	}
	return totEdges;
}

#endif
```

Traversal (Breadth/Depth):

Header File: Sefa_Bujar_HW6_Q3.h:

```
#include "Sefa_B_HW6_Q1.h"
	template<class Item>
	void depthFirst(graph<Item>& G, const size_t& vertex);

	template<class Item>
	void breadthFirst(graph<Item>& G, const size_t& vertex);

	template<class Item>
	void iterateDF(graph<Item>& G, const size_t& vertex, bool* vertexPassed);

#include "Sefa_B_HW6_Q3.cpp"
```

Implementation File: Sefa_Bujar_HW6_Q3.cpp:

```
#include "Sefa_B_HW6_Q1.h"
#include <queue>

template<class Item>
void depthFirst(graph<Item>& G, const size_t& vertex){    assert(vertex < G.size());
	bool * vertexPassed = new bool[G.size()];
	iterateDF(G, vertex, vertexPassed);
}

template<class Item>
void iterateDF(graph<Item>& G, const size_t& vertex, bool* vertexPassed){//(int * neighbors,
const Item& vertex, bool * vertexPassed){
	size_t numNeighbors = 0;
	int * vertexNeighbors = G.neighbors(vertex, numNeighbors);
	bool unprocessedNeighbors = false;
	for(size_t i = 0; i < numNeighbors; i++){
		if(vertexPassed[vertexNeighbors[i]] == false)
			unprocessedNeighbors = true;
	}
	vertexPassed[vertex] = true;
	if(numNeighbors == 0 || !unprocessedNeighbors){
		vertexPassed[vertex] = true;
		return;
	}

	Item nextVertex = 0; /*Will hold the next vertex that we need to iterate through*/
	for(size_t i = 0; i < numNeighbors; i++){
		if(vertexPassed[vertexNeighbors[i]] == false){
			iterateDF(G, vertexNeighbors[i], vertexPassed);
		}
	}
}

template<class Item>
void breadthFirst(graph<Item>& G, const size_t& vertex){
	assert(vertex < G.size());/*First check that we have a valid vertex. */
	queue<Item> gQueue;
	gQueue.push(vertex);
	bool * vertexPassed = new bool[G.size()];
	size_t numNeighbors = 0;
	Item * vertexNeighbors;
	Item currentV;
	/*As long as we still have more neighbors to visit, */
	while(!gQueue.empty()){
		currentV = gQueue.front(); /*store the first guy.*/
		cout << currentV << endl; /*Print the order in which we visit the nodes.*/
		gQueue.pop();
		vertexPassed[currentV] = true; /*Mark the fact that we traverse the current
vertex. */
		/*Find the neightbors*/
		vertexNeighbors = G.neighbors(currentV, numNeighbors);
		for(size_t i = 0; i < numNeighbors; i++){
```

```
                        /*If we didnt pass that neighbor, then push it onto the queue to be
visited. */
                        if(vertexPassed[vertexNeighbors[i]] == false){
                                vertexPassed[vertexNeighbors[i]] = true; /*Mark that you ment
through this neightbor, otherwise its going to double go through them*/
                                gQueue.push(vertexNeighbors[i]);
                        }
                }
        }

}
```

Dijkstra's Algorithm:

Header File: Sefa_Bujar_HW6_Q4.h:

```
#include "Sefa_B_HW6_Q1.h"
#include <queue>
#include <functional>
#include <limits.h>

template<class Item>
void dijkstra(graph<Item> G, const Item& startVertex);


template<class Item>
void relax(graph<int> g, const size_t& indexSource, const size_t& indexTarget, int
distance[], int path[]);

template<class Item>
int indexNextMin(graph<Item> g, int distance[], bool marked[]);

template<class Item>
void initializeSingleSource(graph<Item> g, int distance[], bool marked[], int path[]);

#include "Sefa_B_HW6_Q4.cpp"
```

Implementation File: Sefa_Bujar_HW6_Q4.cpp:

```
#include "Sefa_B_HW6_Q1.h"
#include <queue>
#include <functional>
#include <limits.h>
template<class Item>
void dijkstra(graph<Item> g, const Item& startVertex){
	int distance[g.size()];
	bool marked[g.size()];	int path[g.size()];
	initializeSingleSource(g, startVertex, distance, marked, path);

	for(size_t i = 0; i < g.size(); i++){
		if(marked[i] == false){
		int index = indexNextMin(g, distance, marked);
		marked[index] = true; /*State that we have now gotten through this vertex*/
		if(distance[index] != INT_MAX){
			size_t numNeighbors = 0;
			int * neighborsIndex = g.neighbors(index, numNeighbors);
			for(size_t j = 0; j < numNeighbors; j++){
				if(marked[neighborsIndex[j]] == false){
					relax(g, index, neighborsIndex[j], distance, path);
				}
			}
		}
		}
	}

	/*Printing the arrays to see what the functions do. */
	for(size_t i = 0; i < g.size(); i++){
		cout << "DISTANCE " << distance[i] << " ";
		cout << "Marked " << marked[i] << " ";
		cout << "Path " << path[i] << " ";
		cout << endl;
	}

}

template<class Item>
void relax(graph<Item> g, const size_t& indexSource, const size_t& indexTarget, int
distance[], int path[]){
	int weightSourceTarget  = g.getWeight(indexSource, indexTarget) +
distance[indexSource];
	if(distance[indexTarget] > (weightSourceTarget)){
		distance[indexTarget] = weightSourceTarget;
		path[indexTarget] = indexSource; /*Keep track of which index got us there.....*/
	}
}


template<class Item>
int indexNextMin(graph<Item> g, int distance[], bool marked[]){
	int smallestDistance = INT_MAX;
	int index = -1;
```

```
	for(size_t i = 0; i < g.size(); i++){
		if((distance[i] <= smallestDistance) &&  (marked[i] == false)){
			smallestDistance = distance[i];
			index = i;
		}
	}
	return index;

}



/*Function to initialize all initial values*/
template<class Item>
void initializeSingleSource(graph<Item> g, const Item& startVertex, int distance[], bool
marked[], int path[]){
	for(size_t i = 0; i < g.size(); i++){
		distance[i] = INT_MAX;
		marked[i] = false;
		path[i] = -1;
	}
	distance[startVertex] = 0;
	path[startVertex] = 0; /*because only 0 can actually get there.*/
}
```

Helper: Heapsort (Used for neighbors) Note this heapsort is altered...

Implementation File: Sefa_Bujar_HW5_Q5b.cpp:

```
#include <cstdlib>
#include <cassert>
#include <iostream>

using namespace std;

#include "Sefa_B_HW6_Q1.h"
/*NOTE 11/30/18. These functions have been alterated to heapsort an array based on weights not
just vertices */

void makeHeap(int arr[], int weigh[], size_t size){
	size_t indexNew;
	for(size_t i = 1; i < size; i++){
		indexNew = i;
		while((indexNew>0) && (weigh[indexNew] > weigh[(indexNew-1)/2])){
			int tempArr = arr[indexNew];
			int tempWeigh = weigh[indexNew];
			weigh[indexNew] = weigh[(indexNew-1)/2];
			weigh[(indexNew-1)/2] = tempWeigh;
			arr[indexNew] = arr[(indexNew-1)/2];
			arr[(indexNew-1)/2] = tempArr;
			indexNew = (indexNew-1)/2;
		}
	}
}

void reheapify_down(int arr[], int weigh[], size_t size){
	size_t i = 0;
	if(size == 1){
		return;
	}
	while((i<size) && (2*i+1)<size){
		int tempWeigh = weigh[i];
		int tempArr = arr[i];
		if((2*i+2) >= size){ /*NOTE if our right side is too big, stop this loop.*/
			i = size;
		}
		else if((weigh[i] < weigh[2*i+1])&&(weigh[(2*i+1)]>=weigh[2*i+2])){
			arr[i] = arr[2*i+1];
			arr[2*i+1] = tempArr;
			weigh[i] = weigh[2*i+1];
			weigh[2*i+1] = tempWeigh;
			i = 2*i+1;
		}
		else if((arr[i]< arr[2*i+2]) && (arr[2*i+2] > arr[2*i+1])){
			arr[i] = arr[2*i+2];
			arr[2*i+2] = tempArr;
			weigh[i] = weigh[2*i+2];
			weigh[2*i+2] = tempWeigh;
			i = 2*i+2;
		}
```

```
                else{
                        i = size;
                }
        }

}



void heapSort(int arr[], int weigh[], size_t size){
        size_t i;
        makeHeap(arr, weigh, size); /*Turn array into heap.*/
        i = size;
        while(i > 1){
                --i; /*Decrease index so its valid. Start from last.*/
                /*Swap root with last. */
                int tempArr = arr[0];
                int tempWeigh = weigh[0];
                arr[0] = arr[i];
                arr[i] = tempArr;
                weigh[0] = weigh[i];
                weigh[i] = tempWeigh;
                reheapify_down(arr, weigh, i);
                /*Make sure max is always at the top by rearragning.*/
        }

}
```

In addition to Heapsort, node2.h/cpp was also implemented from the textbook.
http://www.cs.colorado.edu/~main/chapter6/